Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs

Min Yeol Lim* Vincent W. Freeh* David K. Lowenthal†

Abstract

Although users of high-performance computing are most interested in raw performance, both energy and power consumption have become critical concerns. Some microprocessors allow frequency and voltage scaling, which enables a system to reduce CPU performance and power when the CPU is not on the critical path. When properly directed, such dynamic frequency and voltage scaling can produce significant energy savings with little performance penalty.

This paper presents an MPI runtime system that dynamically reduces CPU performance during communication phases in MPI programs. It dynamically identifies such phases and, without profiling or training, selects the CPU frequency in order to minimize energy-delay product. All analysis and subsequent frequency and voltage scaling is within MPI and so is entirely transparent to the application. This means that the large number of existing MPI programs, as well as new ones being developed, can use our system without modification. Results show that the average reduction in energy-delay product over the NAS benchmark suite is 10%—the average energy reduction is 12% while the average execution time increase is only 2.1%.

1 Introduction

High-performance computing (HPC) tends to push performance at all costs. Unfortunately, the "last drop" of performance tends to be the most expensive. One reason is the cost of power consumption, because power is proportional to the product of the frequency and the square of the voltage. As an example of the problem that is faced, several years ago it was observed that on their current trend, the power density of a microprocessor will reach that of a nuclear reactor by the year 2010 [17].

To balance the concerns of power and performance, new architectures have aggressive power controls. One common mechanism on newer microprocessors is the ability of the application or operating system to select the frequency and voltage on the fly. We call this *dynamic voltage and frequency scaling* (DVFS) and denote each possible combination of voltage and frequency a processor state, or *p-state*.

While changing p-states has broad utility, including extending battery life in small devices, the primary benefit of DVFS for HPC occurs when the p-state is reduced in regions where the CPU is not on the critical path. In such a case, power consumption will be reduced with little or no reduction in end-user performance. Previously, p-state reduction has been studied in code regions where the bottleneck is in the memory system [19, 18, 5, 14, 13] or between nodes with different workloads [21].

In contrast, this paper presents a transparent, adaptive system that reduces the p-state in communication phases—that is, in code regions where, while the CPU is not idle, the executed code is not CPU intensive. Our system is built as two integrated components, the first of which trains the system and the second of which does the actual shifting. We designed several training algorithms that demarcate communication regions. In addition, we use a simple metric—operations per unit time—to determine the proper p-state for each region. Next, we designed the shifting component, which includes the mechanics of reducing the p-state at the start of a region and increasing it at the end.

*Department of Computer Science, North Carolina State University, {mlim,vwfreeh}@ncsu.edu

†Department of Computer Science, The University of Georgia, dkl@cs.uga.edu

Because our system is built strictly with code executed within the PMPI runtime layer, there is no user involvement whatsoever. Thus, the large base of current MPI programs can utilize our technique with both no source code change and no recompilation of the MPI runtime library itself. While we aim our system at communication-intensive codes, no performance degradation will occur for computation-intensive programs.

Results on the NAS benchmark suite show that we achieve up to a 20% reduction in energy-delay product (EDP) compared to an energy-unaware scheme where nodes run as fast as possible. Furthermore, across the entire NAS suite, our algorithm that reduces the p-state in each communication region saved an average of 10% in EDP. This was a significant improvement compared to simply reducing the p-state for each MPI communication call. Also, importantly, this reduction in EDP did not come at a large increase in execution time; the average increase across all benchmarks was 2.1%, and the worst case was only 4.5%.

The rest of this paper is organized as follows. In Section 2, we provide motivation for reducing the p-state during communication regions. Next, Section 3 discusses our implementation, and Section 4 discusses the measured results on our power-scalable cluster. Then, Section 5 describes related work. Finally, Section 6 summarizes and describes future work.

2 Motivation

To be effective, reducing the p-state of the CPU should result in a large energy savings but a small time delay. As Figure 1 shows, MPI calls provide an excellent opportunity to reduct the p-state. Specifically, this figure shows the time and energy as a function of CPU frequency for four common MPI calls at several different sizes. (Function MPI_File_write is included because it is used in BT and it communicates with a remote file system, which is a different action from other MPI communication calls.) For all MPI operations, at 1000 MHz at least 20% energy is saved with a time increase of at most 2.6%. The greatest energy savings is 31% when receiving 2 KB at 1000 MHz. In addition, For the largest data size, the greatest time increase was only 5%. In terms of energy-delay product, the minimum is either 1000 or 800 MHz. Overall, these graphs show that MPI calls represent an opportunity, via CPU scaling, for energy saving with little time penalty.

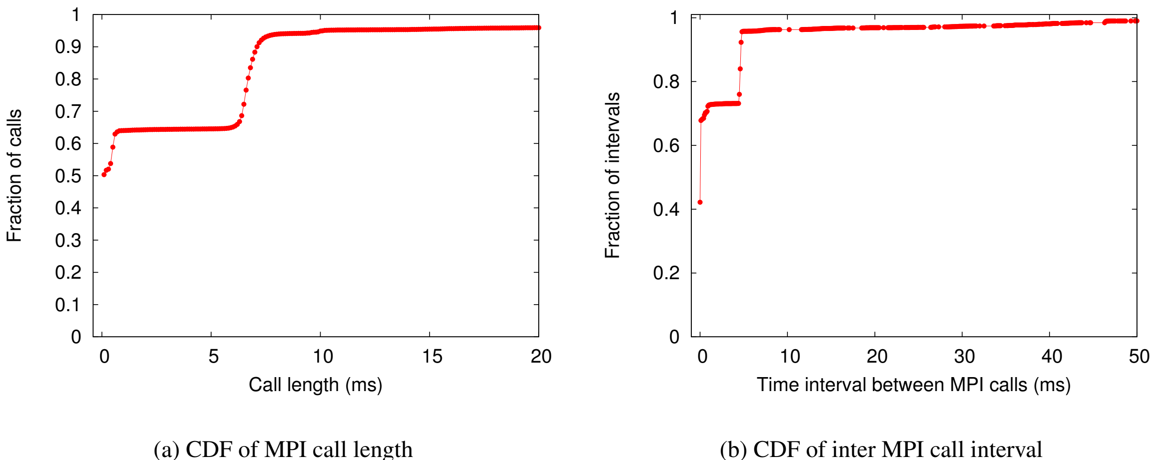
However, in practice, many MPI routines are too short to make reducing the p-state before and increasing the p-state after effective. Figure 2(a) shows a cumulative distribution function (CDF) of elapsed times of all MPI calls in all of the NAS benchmark suite. This figure shows that over 96% of MPI calls take less than 20 ms, and more importantly, 64% take less than 1 ms, and the median value is less than 0.1 ms. Considering that changing the p-state can take up to 700 microseconds, both time *and* energy will increase if one tries to save energy during such short MPI routines. Figure 2(b) plot the CDF for the interval between MPI calls. It shows that 96% of intervals are less than 5 milliseconds, which indicates that MPI calls clustered in time.

Hence, we need to amortize the cost of changing the p-state over several MPI calls. This brings up the problem of how to determine groups of MPI calls, or communication regions, that will execute in the same reduced p-state. The next section describes how we address this problem.

3 Design and Implementation

The overall aim of our system is to save significant energy with at most a small time delay. Broadly speaking, this research has three goals. First, it will identify program regions with a high concentration of MPI calls. Second, it determines the "best" (reduced) p-state to use during such *reducible regions*. Both the first and second goals are accomplished with adaptive *training*. Third, it must satisfy the first two goals with *no involvement by the MPI programmer*, i.e., finding regions as well as determining and shifting p-states should be transparent. In addition to the training component, the system has an *shifting* component. This component will effect the p-state shifts at region boundaries.

While we discuss the training and shifting components separately, it is important to understand that our system does not transition between these components. Instead, program monitoring is continuous,



(a) CDF of MPI call length

(b) CDF of inter MPI call interval

Figure 2: Cumulative distribution functions (CDF) of the duration of and interval between MPI calls for every MPI call for all nine programs in our benchmark suite.

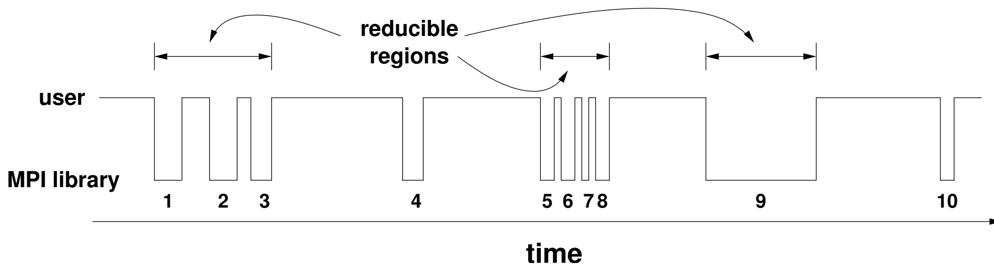


Figure 3: Shows an example trace of an MPI program. The line shows the type of code executed over time. There are 10 MPI calls, and calls 1–3 and 5–8 make up communication regions because they are close enough. Call 9 is considered a region because it is long enough. Even though they are MPI calls, calls 4 and 10 are not in a communication region because they are neither close enough to a call within a region nor long enough in isolation.

and training information is constantly updated. Thus our system is always shifting using the most recent information. Making these components continuous is necessary because we may encounter a given region twice before a different region is encountered.

To achieve transparency—i.e., to implement the training and shifting components without any user code modifications—we use our MPI-jack tool. This tool exploits PMPI [26], the profiling layer of MPI. MPI-jack transparently intercepts (hijacks) any MPI call. A user can execute arbitrary code before and/or after an intercepted call using *pre* and *post* hooks. Our implementation uses the same pre and post hook for all MPI calls.

An MPI routine (such as MPI_Send) can occur in many difference contexts. As a result, the routine itself is not a sufficient identifier. But it is easy to identify the dynamic parent of a call by examining the return program counter in the current activation record. However, because an MPI call can be called from a general utility or wrapper procedure, one dynamic ancestor is not sufficient to distinguish the start or end of a region. Fortunately, it is simple and inexpensive to examine all dynamic parents. We do this by hashing together all the return PCs from all activation records on the stack. Thus, a call is uniquely identified by a hash of all its dynamic ancestors.

3.1 Assumptions

This work is based on two underlying assumptions. First, we assume that the CPU is not on the critical

path during regions of communication, which also implies that it is beneficial to execute in a reduced p-state during this period. Our second assumption is that communication regions have a high concentration of MPI calls. Therefore, we consider two MPI calls invoked in a short period of time to be part of the same communication region.

Figure 3 presents an example trace of an MPI program, where we focus solely on whether the program is in user code or MPI library code. The program begins in user code and invokes 10 MPI calls. In Figure 3 there are two such composite (reducible) regions—$(1, 2, 3)$ and $(5, 6, 7, 8)$. It also has a region consisting of call 9 that is by itself long enough to be executed in a reduce p-state. Thus, Figure 3 has three reducible regions, as well as two single MPI calls that will never execute in a reduce p-state.

3.2 The Training Component

The training component has two parts. The first is distinguishing regions. The second is determining the proper p-state for each region.

Starting from an MPI program with C calls of MPI routines, the goal is to group the C calls into $R \leq C$ regions based on the distance in time between adjacent calls, the time for a call itself, as well as the particular pattern of calls.

First, we must have some notion that MPI calls are close together in time. We denote τ as the value that determines whether two adjacent MPI calls are "close"—anything less than τ indicates they are. Clearly, if τ is ∞, then all calls are deemed to be close together, whereas if it is zero, none are. Next, it is possible that an MPI call itself takes a significant amount of time, such as an MPI_Alltoall using a large array or a blocking MPI_Receive. Therefore, we denote λ as the value that determines whether an MPI call is "long enough"—any single MPI call that executes longer than λ warrants reducing. Our tests use $\tau = 10$ ms and $\lambda = 10$ ms. Section 4.2 discusses the rationale for these thresholds.

Finally, we must choose a training paradigm; we consider three. The first is the degenerate case, *none*, which has no training. It always executes in the top p-state. The next paradigm, *static*, uses only knowledge known prior to execution. This knowledge can be extracted from compiler analysis or execution traces of past runs. The third paradigm is *adaptive*, which is what we focus on in this paper. In this case, the system has no specific knowledge about the particular program, so knowledge is learned and updated as the program executes. For adaptive, there are many way to learn. Next, we discuss two baseline region-finding algorithms, followed by two adaptive region-finding algorithms. Following that, we combine the best region-finding algorithm with automatic detection of the reduced p-state—this serves as the overall training algorithm that we have developed.

First, there is the *base* region-finding algorithm, in which there are no regions—the program always executes in the top p-state. It provides a baseline for comparison. Next, we denote *by-call* as the region-finding algorithm that treats every MPI call as its own region. It can be modeled as $\tau = 0$ and $\lambda = 0$: no calls are close enough and all are long enough. Section 2 explains that *by-call* is not a good *general* solution because the short median length of MPI calls can cause the shifting overhead to dominate. However, in some applications the *by-call* algorithm is very effective, saving energy without having to do any training.

We consider two adaptive region-finding algorithms. The first, called *simple*, predicts each call's behavior based on what it did the last time it was encountered. For example, if a call ended a region last time, it is predicted to end a region the next time it appears. The *simple* method is effective for some benchmark programs. For example consider this pattern.

... AB ... AB ... AB ...

Each letter indicates a particular MPI call invoked from the same call site in the user program; time proceeds from left to right. Variable τ is shown graphically through the use of ellipses, which indicates that the calls are not close. The pattern shows that the distances between A and B are deemed close enough, but the gaps from B to A are not. Thus, the reducible regions in a program with the above pattern always begins with A and ends after B. Because every A begins a region and every B ends a region, once trained, the *simple* mechanism accurately pre-

dicts the reducible region every time thereafter.

However, there exist patterns for which *simple* is quite poor. Consider the following pattern.

... AA ... AA ... AA ...

The gap between MPI calls alternates between less than and greater than τ. However, in this case *both* gaps are associated with the *same* call. Because of this alternation, the *simple* algorithm *always* mispredicts whether A is in a reducible region—that is, in each group *simple* predicts the first A will terminate a region because the second A in the previous group did. The reverse happens when the second A is encountered. This is not a question of insufficient training; *simple* is not capable of distinguishing between the two positions that A can occupy.

Our second region-finding algorithm, *composite*, addresses this problem. At the cost of a slightly longer training phase, *composite* collates information on a *per-region* basis. It then shifts the p-state based on what it learned the last time this region was encountered. It differs from *simple* in that it associates information with regions, not individual calls. The *composite* algorithm matches the pattern of these calls and saves it; when encountered again, it will be able to determine that a region is terminated by the same final call.

The second part of training is determining the p-state in which to execute each region. As mentioned previously, extensive testing indicates that the MPI calls themselves can execute in very low p-states (1000 or 800 MHz) with almost no time penalty. However, a reducible region also contains user code between MPI calls. Consequently, we found that the "best" p-state is application dependent.

The overall adaptive training algorithm we advocate, called *automatic*, uses *composite* to find regions and performance counters to dynamically determine the best p-state on a per-region basis. Specifically, in order to judge the dependence of the application on the CPU in these reducible regions, *automatic* measures the micro-operations[1] retired during the region. It uses the rate micro-operations/microsecond (or OPS) as an indicator of CPU load. Thus a region with a low OPS value is shifted into a low p-state. Our system continually updates the OPS for each region via hardware performance counters. (This means the p-state selection for a region can change over time.) Then, a table developed off line maps OPS to the p-state that will minimize the energy-delay product.

[1]The AMD executes a RISC engine internally. It translates x86 (CISC) instructions into RISC micro-operations. Therefore, micro-operations are a better indicator of CPU performance than instructions.

3.3 The Shifting Component

There are two parts to the shifting component. The first is determining when a program enters and leaves a reducible region. The second part is effecting a p-state shift.

The *simple* region-finding algorithm maintains *begin* and *end* flags for each MPI call (or hash). The pre hook of all MPI calls checks the *begin* flag. If set, this call begins a region, so the p-state is reduced. Similarly, the post hook checks the *end* flag and conditionally resets the top p-state. This flag is updated every time the call is executed: it is set if the region was close enough or long enough and unset otherwise.

Figure 4 shows a state diagram for the *composite* algorithm. There are three states: *OUT*, *IN*, and *RECORDING*. The processor executes in a reduced p-state only in the *IN* state; it always executes in the top p-state in *OUT* and *RECORDING*. The initial state is *OUT*, meaning the program is not in a reducible region. At the beginning of a reducible region, the system enters the *IN* state and shifts to a reduced p-state. The other state, *RECORDING*, is the training state. In this state, the system records a new region pattern.

The system transitions from *OUT* to *IN* when it encounters a call that was previously identified as beginning a reducible region. If the call does not begin such a region but it was close enough to the previous call, the system transitions from *OUT* to *RECORDING*. It begins recording the region from the previous call. It continues recording while calls are close enough.

The system ordinarily transitions from *IN* to *OUT* when it encounters the last call in the region. However, there are two exceptional cases—shown with dashed lines in Figure 4. If the current call is

no longer close enough, then the region is truncated and the state transitions to *OUT*. If the pattern has changed—this current call was not the expected call—the system transitions to *RECORDING*. A new region is created by appending the current call to the prefix of the current region that has been executed. None of the applications examined in Section 4 caused these exceptional transitions.

Our implementation labels each region with the hash of the first MPI call in the region. When in the *OUT* state at the beginning of each MPI call we look for a region with this hash as a label. If the region exists and is marked reducible, then we reduce the p-state and enter the *IN* state.

There are limitations to this approach if a single MPI call begins more than one region. The first problem occurs when some of these regions are marked reducible and some are marked not reducible. Our implementation cannot disambiguate between these regions. The second problem occurs when one reducible region is a prefix of another. Because we do not know which region we are in (long or short), we cannot know whether to end the region after the prefix or not. A more sophisticated algorithm can be implemented that addresses both of these limitations; however, this was not done because it was not necessary for the applications examined. Rather, our system elects to be conservative in time, so our implementation executes in top p-state during all ambiguities. However, this was not evaluated because no benchmark has such ambiguities.

The second part of the execution phase is changing the p-state. In the AMD, the p-state is defined by a frequency identifier (FID) and voltage identifier (VID) pair. The p-state is changed by storing the appropriate FID-VID to the *model specific register* (MSR). Such a store is privileged, so it must be performed by the kernel. The *cpufreq* and *powernow* modules provide the basic support for this. Some additional tool support and significant "tweaking" of the FID-VID pairs was needed to refine the implementation. The overhead of changing the p-state is dominated by the time to scale—not the overhead of the system call. For the processor used in these tests, the upper bound on a p-state transition (including system call overhead) is about 700 microseconds, but the average is less than 300 microseconds [12].

4 Results

This section presents our results in three parts. The first part discusses results on several benchmark programs on a power-scalable cluster. These results were obtained by applying the different algorithms described in the previous section. The second part gives a detailed analysis of several aspects of our system and particular applications.

Benchmark attributes Eight of our benchmark applications come from the NAS parallel benchmark suite, a popular high-performance computing benchmark [3]. The NAS suite consists of scientific benchmarks including application areas such as sorting, spectral transforms, and fluid dynamics. We test class C of these benchmarks. The benchmarks are unmodified with the exception of BT, which took far longer than all the other benchmarks; to reduce the time of BT without affecting the results, we executed it for 60 iterations instead of the original 200. The ninth benchmark is *Aztec* from the ASCI Purple benchmark suite [2].

Table 1 presents overall statistics of the benchmarks running on 8 or 9 nodes, which are distilled from traces that were collected using MPI-jack. The second column shows the number of MPI calls invoked dynamically. The number of regions (column 3) is determined by executing the *composite* algorithm on the traces with the close enough threshold (τ) set to 10 ms. The table shows the average number of MPI calls per region in the fourth column. Because some MPI calls are not in reducible regions, the product of the third and fourth columns is not necessarily equal to the second column. Next, the table shows the average time per MPI call and per reducible region. The last two columns show the fraction of the overall time spent in MPI calls and regions, respectively

The table clearly shows that these applications have diverse characteristics. In particular, in terms of the key region parameters—the number of regions, the number of calls per region, and the duration of the regions—there are at least two orders of magnitude difference between the greatest and least values. Finally, with the exception of EP, the reducible fraction is at least 44%.

	MPI calls	Reducible regions	Calls per region	Average time (ms)		Time fraction	
				Per MPI	Per region	MPI	region
EP	5	1	4	68.7	337.0	0.005	0.005
FT	46	45	1.0	18,400	18,810	0.849	0.860
IS	37	14	2.5	3100	8200	0.871	0.871
Aztec	20,767	301	68.9	2.04	143	0.806	0.812
CG	41,953	1977	21.2	6.90	149	0.753	0.768
MG	10,002	158	63.3	3.77	272	0.500	0.574
SP	19,671	8424	3.2	20.6	49.4	0.441	0.453
BT	108,706	797	136.7	8.39	1145	0.865	0.891
LU	81,874	766	107.2	1.11	356	0.149	0.446

Table 1: Benchmark attributes. Region information from *composite* with $\tau = 10$ ms.

As we will show below, while several of the particular algorithms we developed may be best for *some* benchmark, the wide variety of characteristics in the entire suite means that they may be quite poor on other benchmarks. Therefore, our goal was to find an algorithm that works *well* for *all* benchmarks. Below we will show that the *automatic* algorithm is such an algorithm.

Experimental methodology For all experiments, we used a 10-node AMD Athlon-64 cluster connected by a 100Mbps network. Each node has 1 GB of main memory. The Athlon-64 CPU supports 7 p-states (from 2000 to 800 MHz). Each node runs the Fedora Core 3 OS and Linux kernel 2.6.8, and frequency shifting was done through the *sysfs* interface. All applications were compiled with either *gcc* or the Intel Fortran compiler, using the O2 optimization flag. We controlled the entire cluster, so all experiments were run when the only other processes on the machines were daemons.

Elapsed time measurements are from the system clock using the *gettimeofday* system call. A power meter sits in between the system power supply of each node and the wall. We read this meter to get the power consumption. These readings are integrated over time to yield energy consumption. Thus the energy consumption measurements are empirically gathered for the system.

4.1 Overall Results

Our work trades time for energy. This tradeoff is difficult to evaluate because it has two dimensions, and users may assign different values to the tradeoff. Energy-delay product (EDP) is one canonical evaluation metric. It is used here to convert the two-dimensional energy-time tradeoff into a single scalar metric. Figure 5 shows the EDP of each method relative to the base case. This is a subset of the data in Table 2. We conducted five tests for each of the benchmarks. For each test, the measured elapsed time and consumed energy are presented. EDP is computed from these empirical measurements. The value relative to the base case is also presented. All tests were conducted a minimum of three times, with the median value reported. The variance between any two runs was very small, so it is not reported.

As mentioned previously, the *base* test executes the program solely in top frequency (2000 MHz). Linux puts the processor into a low-power halt state when the run queue is empty, so we note that the base test is already power-efficient to some degree. Recall that the *by-call*, *simple*, and *composite* are purely region-finding algorithms and do not choose a particular p-state for regions. To provide a comparison with *automatic*, which uses OPS to choose the p-state on the fly, we did an exhaustive search of p-states. Then, the results for *by-call*, *simple* and *composite* reported in Figure 5 and Table 2 are those using the p-state yielding the lowest EDP.

The *automatic* test shows the results of the algorithm that identifies regions with *composite* and se-

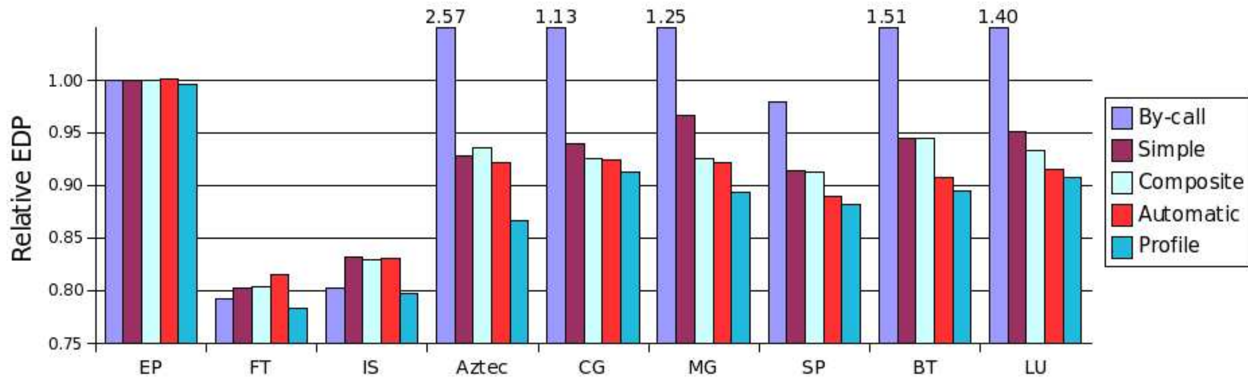


Figure 5: Overall EDP results for 5 methods relative to base case. For readability, the y-axis starts at 0.75.

lects the reduced p-state for each region using OPS. The last column in Table 2 shows the results of *profile*; it executes a schedule of p-state changes that were produced from profiling traces. Essentially, it is *automatic* without any mispredictions. The *profile* tests represents the "best one can do," in that uses our best algorithm (*automatic*) with prior run information. The first three algorithms use a single reduced gear throughout, while the last two may use several reduced gears depending on the OPS.

The applications fall into several categories based on the performance of the algorithms. EP executes less than 0.5% of the total execution time in MPI calls, so no energy savings is possible with our mechanisms. Thus we do not evaluate EP any further.

The second category contains FT and IS, applications that perform well with the trivial *by-call* method. These are the only two applications that have singleton regions—an individual MPI call that is longer than λ, thus allowing a reduction in the p-state. In FT, no calls are within τ. The average length of MPI calls is 18.4 and 3.1 seconds for FT and IS, respectively. Consequently, the reducible regions are almost entirely MPI code and the reduced p-state is 800 MHz—the lowest p-state. As a result, these two benchmarks have the greatest energy saving. Although these programs are in the reduced p-state more than 85% of the time, the increase in execution time is less than 1%. Therefore, the EDP is very low. This is our best case in terms of EDP, but most applications do not fall into this category. For example, SP is the only other benchmark that has an energy savings using *by-call* because it has a relatively large average time per MPI call.

The next class of programs are ones for which *by-call* performs poorly due to excessive p-state switching, but *simple* is adequate. In our benchmark suite, this includes only Aztec. The pattern of MPI calls in Aztec is such that *simple* predicts well so there are few false positives for *composite* to remove. Moreover, the longer training required by *composite* creates false negatives—which are lost opportunities to save energy—that do not occur in *simple*. Therefore, *composite* actually spends more time in the wrong p-state than *simple*. As a result, *simple* is 0.7% lower in EDP, 1% less energy but 0.5% more time, than *composite*.

The fourth category, which consists of five programs, are those that both *by-call* and *simple* do not perform well. In CG and MG, the EDP of *composite* is lower than *simple* because of improved region finding. In these benchmarks, *composite* eliminates both excessive p-state switching of *by-call* as well as a non-trivial number of false predictions of *simple*. Moreover, there is little difference between *composite* and *automatic*. On the other hand, there is no difference between the region finding of *simple* and *composite* in BT and SP. The *automatic* algorithm has a lower EDP than *simple* or *composite* because it is not limited to selecting a single reduced p-state. The improvement of *automatic* is due to the dynamic selection of a reduced p-state for each region. LU benefits from both improved region finding and multiple p-state selection.

FT is the only application for which *automatic* is not as good as *composite*. The reason is that the

		Base	By-call		Simple		Composite		Automatic		Profile	
	Time (s)	75.53	75.55	1.000	75.52	1.000	75.47	0.999	75.45	0.999	75.49	0.999
EP	**Energy** (KJ)	59.876	59.878	1.000	59.857	1.000	59.878	1.000	59.946	1.001	59.669	0.997
	EDP (MJ·s)	4.5225	4.5238	1.000	4.5202	1.000	4.5193	1.000	4.5289	1.001	4.5046	0.996
	Time (s)	984.85	987.24	1.002	985.80	1.001	985.36	1.001	983.99	1.000	985.99	1.001
FT	**Energy** (KJ)	606.45	479.24	0.790	486.21	1.802	487.41	0.804	494.91	0.816	475.13	0.783
	EDP (MJ·s)	597.26	473.12	0.792	479.30	0.802	480.27	0.804	486.99	0.815	468.48	0.784
	Time (s)	133.26	133.87	1.005	133.41	1.001	133.48	1.002	133.51	1.002	133.70	1.003
IS	**Energy** (KJ)	79.102	63.236	0.799	65.768	0.831	65.465	0.828	65.604	0.829	62.891	0.795
	EDP (MJ·s)	10.541	8.4655	0.803	8.7739	0.832	8.7379	0.829	8.7590	0.831	8.4086	0.798
	Time (s)	51.94	85.98	1.655	55.80	1.074	55.55	1.069	53.55	1.031	53.86	1.037
Aztec	**Energy** (KJ)	31.945	49.571	1.552	27.589	0.864	27.923	0.874	28.578	0.895	26.670	0.835
	EDP (MJ·s)	1.6593	4.2619	2.568	1.5394	0.928	1.5510	0.935	1.5305	0.922	1.4364	0.866
	Time (s)	378.50	414.87	1.096	402.81	1.064	396.74	1.048	393.49	1.039	396.14	1.047
CG	**Energy** (KJ)	238.58	245.85	1.031	210.52	0.882	209.63	0.879	212.16	0.889	207.88	0.871
	EDP (MJ·s)	90.301	101.00	1.130	84.801	0.939	83.565	0.925	83.484	0.924	82.348	0.912
	Time (s)	76.84	90.35	1.176	81.90	1.066	78.87	1.027	78.67	1.024	78.88	1.027
MG	**Energy** (KJ)	55.173	58.631	1.063	49.980	0.906	49.705	0.901	49.639	0.900	48.058	0.871
	EDP (MJ·s)	4.2393	5.2971	1.250	4.0933	0.966	3.9204	0.925	3.9051	0.921	3.7909	0.894
	Time (s)	910.85	944.86	1.037	938.16	1.030	938.38	1.030	945.40	1.038	932.00	1.023
SP	**Energy** (KJ)	758.56	715.92	0.944	672.85	0.887	672.01	0.886	650.80	0.858	653.86	0.862
	EDP (MJ·s)	690.94	676.44	0.979	631.24	0.914	630.60	0.913	615.27	0.890	609.42	0.882
	Time (s)	1027.2	1295.5	1.261	1061.3	1.033	1057.1	1.029	1040.3	1.013	1029.3	1.002
BT	**Energy** (KJ)	646.01	774.96	1.200	590.38	0.914	592.93	0.918	579.35	0.897	577.01	0.893
	EDP (MJ·s)	663.58	1003.94	1.513	626.54	0.944	626.80	0.945	602.67	0.908	593.92	0.895
	Time (s)	628.74	841.69	1.339	662.94	1.054	654.67	1.041	657.12	1.045	658.83	1.048
LU	**Energy** (KJ)	489.08	510.24	1.043	441.24	0.902	438.14	0.896	428.24	0.876	423.19	0.865
	EDP (MJ·s)	307.51	429.46	1.397	292.51	0.951	286.83	0.933	281.41	0.915	278.81	0.907

Table 2: Overall performance of benchmark programs. Where appropriate, the value relative to the base case is also reported. The p-state used in *simple* and *composite* is the one found to have the *best* energy-delay product.

region characteristics change over time. According to OPS, the best gear for the primary region should be 1200 MHz for the first occurrence and 800 MHz thereafter. The *composite* algorithm uses 800 MHz for all, *automatic* first uses 2000 MHz, then 1200 MHz, and not until the third occurrence does it select 800 MHz. This delay in determining the best p-state results in a 1.1% higher EDP in *automatic* than in *composite*. Again, keep in mind that *composite* (and *simple* and *by-call*) is using the p-state that was best over all possibilities.

Finally, the rightmost column shows the performance of *profile*. In all cases, *profile* has the best EDP; this is expected because it uses prior application knowledge (a separate profiling execution). The usefulness of *profile* is that it serves as a rough lower bound on EDP. It is not a precise lower bound because of several reasons, including that the reduced p-state in a given region cannot be proven to be best for EDP. However, generally speaking, if a given algorithm results in EDP that is close to *profile*, it is successful.

Summary In summary, *simple* is generally better than *by-call* due to less p-state switching overhead. However, *composite* is generally better than *simple* because it has, for several benchmarks, far fewer false positives at the cost of a a small number of additional false negatives. Finally, *automatic* is better than *simple* or *composite* in all programs except for FT (IS is within experimental error), because using a customized p-state for each region is better than a single p-state—even the best one possible based on exhaustive testing—for the whole program.

4.2 Detailed Analysis

System Overhead This paragraph discusses the overhead of the system in two parts. The first part is the overhead of changing p-states. As stated above on this AMD-64 microprocessor the average cost to shift between p-states is slightly less than 300 microseconds. The greatest time to shift is just under 700 microseconds [12]. The actual cost of this overhead depends on the number of shifts between p-states, which is twice the number of reducible re-

gions. Table 1 shows that the shortest average region length is approximately 50 milliseconds, which is more than 160 times longer than the average shift cost. Thus the overhead due to shifting is nominal.

The second part of the overhead is due to data collection, which happens at the beginning and end of each MPI call. The overhead consists of hijacking the call, reading the time, and the performance counters. The cost of each data collection is approximately ZZZ microseconds. The relative cost of this overhead is less than 1% on all the NAS programs. The overhead on Aztec is higher because has 1 or more orders of magnitude more MPI operations per second than the NAS benchmark. Nevertheless, it is only Y%.

This section focuses on the details of how we achieved the performance reported in the previous section. In particular, we first account false negatives and positives. Then, we investigate how we selected our threshold values.

False negatives and positives Figure 6 compares *simple* and *composite*. The algorithms are applied postmortem to the collected trace data. In practice, reducing the p-state can increase time. Furthermore, there is overhead in shifting p-states and executing each algorithm. This analysis does not consider either of these factors because adding such costs analytically is approximate at best. Rather, this analysis shows the *potential* impact by showing how much of the base code is effected by our scaling.

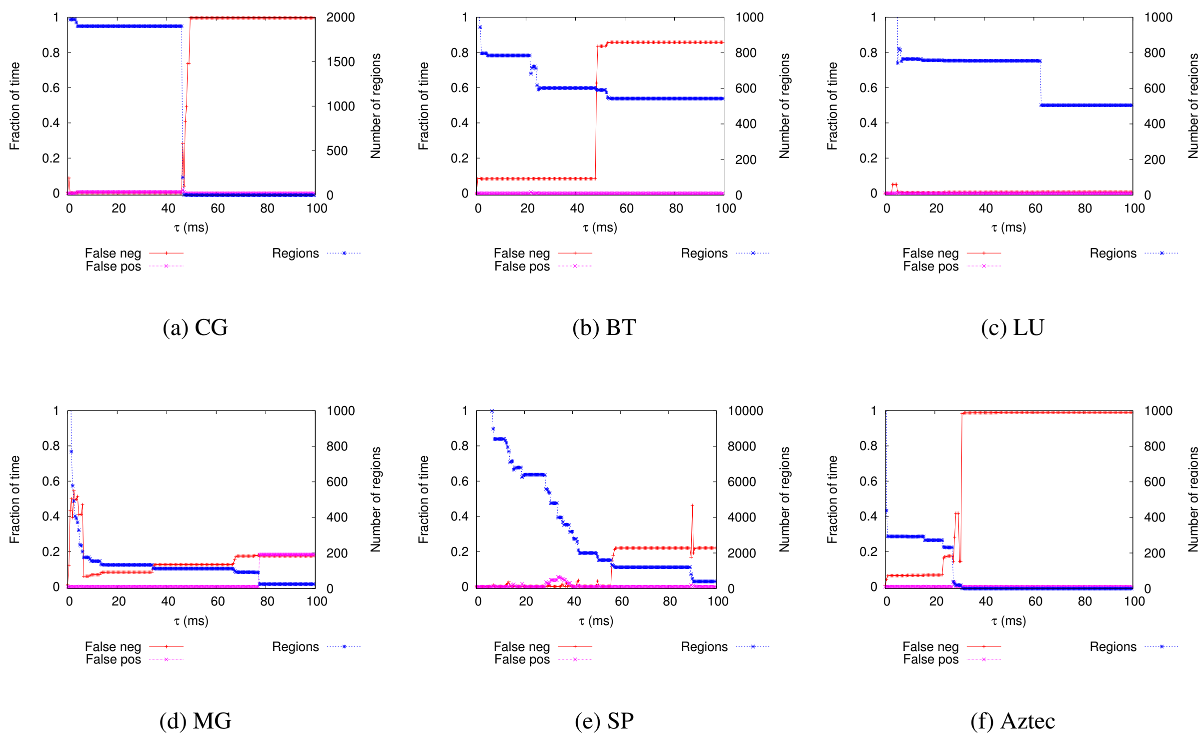
This analysis divides the base, unscaled time into into four categories along two dimensions: (a) *IN* or *OUT* of a reducible region and (b) *true* or *false* prediction. Correct predictions are determined by examining trace data, as is done in *profile*. In Figure 6, bars labeled "IN (true)" and "OUT (true)" are correctly predicted. The bar labeled "IN (false)" is a false positive; the application mistakenly executes in a reduced p-state instead of the top p-state. The bar labeled "OUT (false)" is a false negative; the application mistakenly executes in the top p-state instead of a reduce one. Generally, a false positive is considered worse than a false negative because it can result in a time penalty. In contrast, a false negative represents an energy saving opportunity lost.

For FT and IS, *simple* has false negatives, which explains why it is worse in EDP than *by-call*. Thus, *simple* is slightly faster but consumes more energy. Furthermore, with singleton regions there is no difference in the predictive power of *simple* and *composite*. These plots show that the longer training period in *composite* adds false negatives to Aztec and BT. But because neither application has significant false positives to eliminate, *simple* performs better than *composite*. For the remaining three benchmarks, *composite* eliminates significant mispredictions compared to *simple*. For CG and LU there are in fact no mispredictions in *composite*. For MG, all false positive cases are eliminated as well.

Choosing τ Figure 7 shows data for evaluating τ, the threshold that determines whether two MPI routines are "close enough." This evaluation is done with *composite*. Three metrics are plotted: the fraction of time for false negatives, the fraction of time for false positives, and the number of regions.

Clearly, there is a tension in the choice of τ. If τ is too large, then a few, large regions will result. This means that when using *automatic*, a large number of false negatives would result, which in turn would result in a significant fraction of time due to them. For example, if $\tau = \infty$, the training component will run the whole program, and the executing component will never run—the entire program is a false negative. On the other hand, if τ is too small, then *automatic* would approach *by-call*, which we know from Section 4.1 is often an ineffective algorithm due to excessive time due to p-state shifting.

Indeed, the figure shows that the value of τ has a significant impact on benchmarks that have several short MPI calls (EP, IS, and FT do not and so are not in the figure). The goal is, essentially, to choose for τ the smallest value possible such that the region overhead (which is proportional to the number of regions) is small. While there is not a choice that works for all benchmarks—for CG it is around 45 ms, for BT 48 ms, but for the others, it is closer to 10 or 20 ms. The one problematic application is SP, which has numerous short regions, and the choice of a longer τ can reduce EDP by more than 2%. We are currently addressing this issue.



(a) CG (b) BT (c) LU

(d) MG (e) SP (f) Aztec

Figure 7: Evaluating τ in *composite*. Show fraction of time mispredicted and number of regions (right-hand y-axis) as a function of τ. For readability, right-hand y-axis has different scale for CG and SP.

Choosing λ The duration of MPI calls over all benchmarks is generally small—Section 2 showed that 95% of the calls take less than 10 ms. Basically, there is an extreme bimodal distribution of times for MPI calls over the programs, because the calls in IS and FT take 3 and 18 *seconds*, respectively. Thus all values of λ between 10 and 2000 ms achieve the same result. We do point out, though, that for different benchmarks, where the calls may be more evenly distributed, much more thought would need to go into choosing λ.

5 Related Work

The most relevant related work to this paper is in high-performance, power-aware computing. Several researchers have addressed saving energy with minimal performance degradation. Cameron *et al.* [5] uses a variety of different DVS scheduling strategies (for example, both with and without application-specific knowledge) to save energy without significantly increasing execution time. A similar run-time effort is due to Hsu and Feng [18]. Our own prior work is fourfold: an evaluation-based study that focused on exploring the energy/time tradeoff in the NAS suite [14], development of an algorithm for switching p-states dynamically between phases [13], leveraging load imbalance to save energy [21], and minimizing execution time subject to a cluster energy constraint [29].

The difference between all of the above research and this paper is the type of bottleneck we are attacking. This is the first work we know of to address the communication bottleneck.

The above approaches strive to save energy for a broad class of scientific applications. Another approach is to save energy in an application-specific way; the work in [8] used this approach for a parallel sparse matrix application.

There are also a few high-performance computing clusters designed with energy in mind. One is BlueGene/L [1], which uses a “system on a chip” to re-

duce energy. Another is Green Destiny [30], which uses low-power Transmeta nodes. Unlike our approach, these machines use less powerful processors.

Another area of work closely aligned with saving energy in HPC applications is saving energy in server systems. In sites such as hosting centers where there is a sufficiently large number of machines, energy management may become an issue; see [7, 25, 11] for examples of this using commercial workloads and web servers. Such work shows that power and energy management are critical for commercial workloads, especially web servers [22]. Additional approaches have been taken to include dynamic voltage scaling (DVS) and request batching [10]. The work in [27] applies real-time techniques to web servers in order to conserve energy while maintaining quality of service.

In server farms, disk energy consumption is also significant; several have studied reducing disk energy (*e.g.*, [6, 31, 24]). In this paper, we do not consider disk energy as it is generally less than CPU energy, especially if scientific programs operate primarily in core.

Our work attempts to infer regions based on recognizing repeated execution. Others have had the same goal and carried it out using the program counter. Gniady and Hu used this technique to determine when to power down disks [16], along with buffer caching pattern classification [15] and kernel prefetching [4]. In addition, dynamic techniques have been used to find program phases [20, 9, 28], which is tangentially related to our work.

6 Conclusion

This paper has presented a transparent, adaptive system for reducing the p-state in communication phases. The basic idea is to find such regions on the fly by monitoring MPI calls and keep a state machine that recognizes the regions. Then, we use performance counters to guide our system to choose the best p-state in terms of energy-delay product (EDP). User programs can use our runtime system with zero user involvement. Results on the NAS benchmark suite showed an average 10% reduction in EDP across the NAS suite.

Our future plans include integrating the idea of reducing the p-state during communication regions with our past work on reducing the p-state during computation regions. Additionally, we will conduct these experiments on a newer cluster, which has a gigabit network and multi-core, multiprocessor nodes. For computation, we leverage the memory or node bottleneck to save energy, sometimes with no increase in execution time. The overall goal is to design and implement one MPI runtime system that simultaneously exploits all three bottlenecks.

References

[1] N.D. Adiga et al. An overview of the BlueGene/L supercomputer. In *Supercomputing*, November 2002.

[2] ASCI Purple Benchmark Suite. http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks/.

[3] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. RNR-91-002, NASA Ames Research Center, August 1991.

[4] Ali Raza Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *SIGMETRICS*, pages 157–168, 2005.

[5] K.W. Cameron, X. Feng, and R. Ge. Performance-constrained, distributed dvs scheduling for scientific applications on power-aware clusters. In *Supercomputing*, November 2005.

[6] Enrique V. Carrera, Eduardo Pinheiro, and Ricardo Bianchini. Conserving disk energy in network servers. In *Intl. Conference on Supercomputing*, June 2003.

[7] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centres. In *Symposium on Operating Systems Principles*, 2001.

[8] Guilin Chen, Konrad Malkowski, Mahmut Kandemir, and Padma Raghavan. Reducing power with performance contraints for parallel sparse applications. In *Workshop on High-Performance, Power-Aware Computing*, April 2005.

[9] A. Dhodapkar and J. Smith. Comparing phase detection techniques. In *International Symposium on Microarchitecture*, pages 217–227, December 2003.

[10] Elmootazbellah Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy conservation policies for web servers. In *Usenix Symposium on Internet Technologies and Systems*, 2003.

[11] E.N. (Mootaz) Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *Workshop on Mobile Computing Systems and Applications*, Feb 2002.

[12] Mark E. Femal. Non-uniform power distribution in data centers for safely overprovisioning circuit capacity and booasting throughput. Master's thesis, North Carolina State University, Raleigh, NC, May 2005.

[13] Vincent W. Freeh, David K. Lowenthal, Feng Pan, and Nandani Kappiah. Using multiple energy gears in MPI programs on a power-scalable cluster. In *Principles and Practices of Parallel Programming*, June 2005.

[14] Vincent W. Freeh, David K. Lowenthal, Rob Springer, Feng Pan, and Nandani Kappiah. Exploring the energy-time tradeoff in MPI programs on a power-scalable cluster. In *International Parallel and Distributed Processing Symposium*, April 2005.

[15] Chris Gniady, Ali Raza Butt, and Y. Charlie Hu. Program-counter-based pattern classification in buffer caching. In *OSDI*, pages 395–408, 2004.

[16] Chris Gniady, Y. Charlie Hu, and Yung-Hsiang Lu. Program counter based techniques for dynamic power management. In *HPCA*, pages 24–35, 2004.

[17] Richard Goering. Current physical design tools come up short. *EE Times*, April 14 2000.

[18] Chung hsing Hsu and Wu chun Feng. A power-aware run-time system for high-performance computing. In *Supercomputing*, November 2005.

[19] Chung-Hsing Hsu and Wu-chun Feng. Effective dynamic-voltage scaling through CPU-boundedness detection. In *Fourth IEEE/ACM Workshop on Power-Aware Computing Systems*, December 2004.

[20] M. Huang, J. Renau, and J. Torellas. Positional adaptation of processors: Application to energy reduction. In *International Symposium on Computer Architecture*, June 2003.

[21] Nandani Kappiah, Vincent W. Freeh, and David K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *Supercomputing (to appear)*, November 2005.

[22] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commerical servers. *IEEE Computer*, pages 39–48, December 2003.

[23] Orion Multisystems. http://www.orionmulti.com/.

[24] Athanasios E. Papathanasiou and Michael L. Scott. Energy efficiency through burstiness. In *Workshop on Mobile Computing Systems and Applications*, October 2003.

[25] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on Compilers and Operating Systems for Low Power*, September 2001.

[26] Rolf Rabenseifner. Automatic profiling of MPI applications with hardware performance counters. In *PVM/MPI*, pages 35–42, 1999.

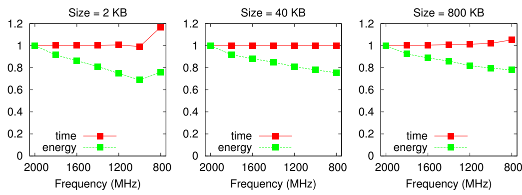
[27] Vivek Sharma, Arun Thomas, Tarek Abdelzaher, and Kevin Skadron. Power-aware QoS management in web servers. In *IEEE Real-Time Systems Symposium*, Cancun, Mexico, December 2003.

[28] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Architectural Support for Programming Languages and Operating Systems*, October 2002.

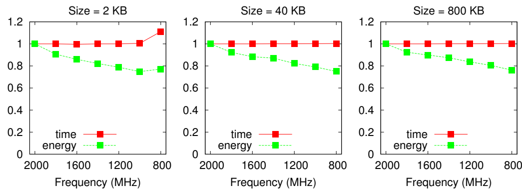
[29] Robert C. Springer IV, David K. Lowenthal, Barry Rountree, and Vincent W. Freeh. Minimizing execution time in MPI programs on an energy-constrained, power-scalable cluster. In *ACM Symposium on Principles and Practice of Parallel Programming*, March 2006.

[30] M. Warren, E. Weigle, and W. Feng. High-density computing: A 240-node beowulf in one cubic meter. In *Supercomputing*, November 2002.

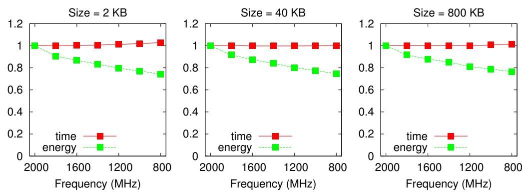
[31] Qingbo Zhu, Francis M. David, Christo Devaraj, Zhenmin Li, Yuanyuan Zhou, and Pei Cao. Reducing energy consumption of disk storage using power-aware cache management. In *High-Performance Computer Architecture*, February 2004.



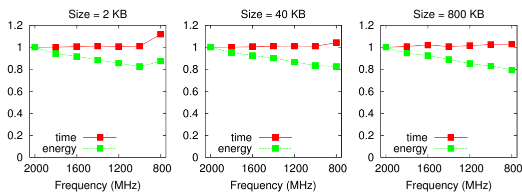
(a) MPI_Recv



(b) MPI_Send



(c) MPI_Alltoall



(d) MPI_File_write_all

Figure 1: Micro-benchmarks showing time and energy performance of MPI calls with CPU scaling.

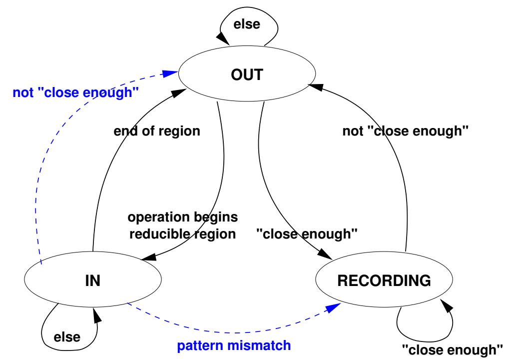
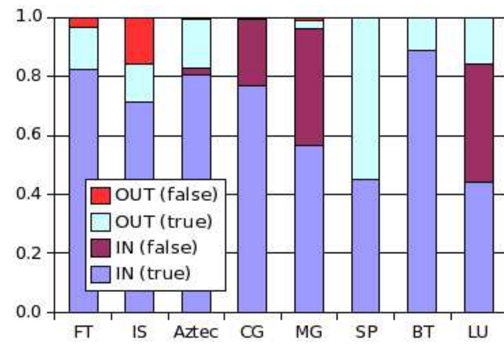
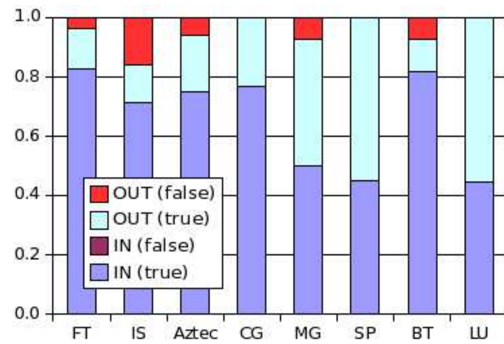


Figure 4: State diagram for *composite* algorithm.



(a) *simple*



(b) *composite*

Figure 6: Breakdown of execution time for adaptive region-finding algorithms.